Are functional languages a good way to represent productive meta–models ?

Sébastien Mosser

University of Nice (France)
CNRS – I3S Lab – Rainbow team

4^{th} European Lisp Workshop, ECOOP'07

Abstract

Following *Model Driven Development* guidelines, developers will define meta–models, models and then implement transformations between models. Existing tools based on models require highly specific skills and knowledge from developers, and use *Domain Specific Language* (Dsl) as the entry point for final users.

Is it possible to describe Dsl–based meta–models using functional programming concepts and languages ? Can we do fast *Model Driven Development* using such techniques ?

1 Introduction : a model world

The *Model Driven Architecture* [OMG, 2005] paradigm is based on *model definitions* and *model transformations* (or refinement). High-level abstract models are progressively transformed into concrete platform models. As an endless hierarchy, each model is described by a *meta–model*, itself a model.

Comparisons between models and grammar have been studied since several years [Klint et al., 2005]. For example, the Lisp language describes Lisp programs, and the compiler will transform it into native code at runtime. Therefore, Lisp language can be considered as a meta–model describing Lisp programs *Platform Independent Model*, and the interpreter implements a transformation from this Pim to native executable code *Platform Specific Model*.

Model–focused researches are based on standards like Uml, Atl, Qvt, ... In [Muller and Hassenforder, 2005], the authors show a bridge between *Domain Specific Languages* (Dsl) based *GrammarWare* and *ModelWare* using those techniques.

The goal of this short paper is to discuss, relying on our experience in definition of meta–models, how functional programming concepts can be used to define Dsl–based meta–models and implement transformations. Section 2 describes a simple meta–model and identifies some transformations. We propose

a functional approach of model definition and transformations in section 3. Section 4 describes a related work developed by our team using this technique, and discusses some limits discovered during this development. Finally, section 5 concludes this paper.

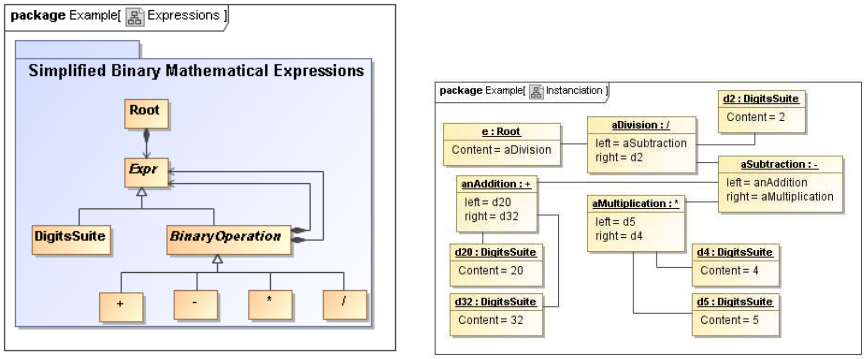
2 A productive meta–model example

We use as an example a simplified meta–model for *arithmetic binary expression*. We inductively define an expression as a number (sequence of digits) or as a binary operator applied to two expressions. A DSL (FIG. 1) support model definition following this meta–model (FIG. 2(a)).

Binary expression meta–model is considered as *productive* regarding possible refinements. An expression can be transformed into several target models. This section focus on two kinds of targets : *(i)* evaluation platforms and *(ii)* display platforms.

$$
\begin{array}{lcl}
\textsc{Root} & ::= & \textsc{<Expr>} \\
\textsc{Expr} & ::= & 0 \mid [1-9]^{+}[0-9]^{*} \mid (\ \textsc{<Expr>}\ ) \\
 & \mid & \textsc{<Expr>}\ +\ \textsc{<Expr>} \mid \textsc{<Expr>}\ -\ \textsc{<Expr>} \\
 & \mid & \textsc{<Expr>}\ *\ \textsc{<Expr>} \mid \textsc{<Expr>}\ /\ \textsc{<Expr>}
\end{array}
$$

Figure 1: Binary expression simplified grammar



(a) UML expressions *meta–model* (b) *Model* example

Figure 2: UML description

Models : Using this DSL, users can express *models*. We consider in this paper as an example the expression e defined as $e = ((20+32)-(5*4))/2$ (UML instantiation is shown in FIG. 2(b)).

Model productions : There are a lot of existent heterogeneous languages able to evaluate a binary expression like e. We can use for example *(i)* `dc` (reversed polish notation), *(ii)* LISP (fully nested and parenthesised notation) or *(iii)* PYTHON (infix notation). TAB. 1 show concrete syntax for e using those languages.

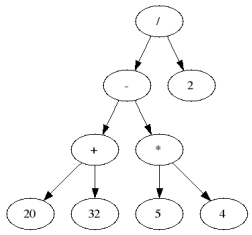
Language	*Concrete Syntax* describing e
`dc`	`dc -e "20 32 + 5 4 * - 2 / p"`
LISP	`(format t "~a~%" (/ (- (+ 20 32) (* 5 4)) 2))`
PYTHON	`print ((20+32) - (5*4)) / 2`

Table 1: Expression evaluation target languages

We could also display mathematical expression using various tools, like LaTeX, `GraphViz`, `MathML`,... as shown in FIG. 3. Instead of the previous ones, those meta–models use a very different basis than the source meta–model.

$$e = \frac{(20+32)-(5\times 4)}{2}$$

(a) LaTeX



(b) `GraphViz`

Figure 3: Expression display target languages

Productive meta–models implementation : There are several usual approaches of productive meta–model implementation. A first way is to consider models as objects graphs and use *design patterns* [Gamma et al., 1993] to implement transformations (as a *visit* of the object graph). Another way is to implement models and transformations as in [Muller et al., 2005], using specialised environment.

Problems : Those techniques are heavyweight. To implement expected transformations, you will have to formally define source and targets meta–model elements, using specialised tools or "*good practices*". Even in front of a simple

problem (*e.g.* transform an abstract arithmetic expression into several targets in a extensible way), the initial problem seem overwhelmed by those tools.

3 Model transformations as s–expr evaluation

Models & Transformations : We propose to use functional languages *s–expression*s to represent a productive meta–model. The DSL compiler will produce unevaluated *s–expression*s instead of objects graphs.

As models are defined as *s–expression*s, models transformations or refinements can be implemented using native LISP *function definitions* mechanisms. A transformation is then implemented as a set of function definitions, following meta–model definition and targeted platform specificities.

Example : We define in the following code (LISTING. 1, *line* 2) an *s–expression*[1] for e , and two possible transformations. The first one produces interpretable LISP code, and the second one produces `dc` command line invocation. The project website (*cf.* section 4) shows more complex transformations based on another meta–model.

```
1  ;; e <- ((20 + 32) - (5 * 4)) / 2
   (setq e (quote (root (divide (minus (plus 20 32) (star 5 4) 2)))))
3
   ;; Example  #1 : s-expr -> Lisp code
5  (defun root   (exp) (format t "~a␣=␣~a~%" exp (eval exp)))
   (defun divide (a b) (list (quote /) a b))
7  (defun minus  (a b) (list (quote -) a b))
   (defun plus   (a b) (list (quote +) a b))
9  (defun star   (a b) (list (quote *) a b))
   (eval e)
11
   ;; Example #2 : s-expr ->  dc command line
13 (defun root   (exp) (format t "dc␣-e␣\"~a␣p\"␣~%" exp))
   (defun divide (a b) (format nil "~a␣~a␣/␣" a b))
15 (defun minus  (a b) (format nil "~a␣~a␣-␣" a b))
   (defun plus   (a b) (format nil "~a␣~a␣+␣" a b))
17 (defun star   (a b) (format nil "~a␣~a␣*␣" a b))
   (eval e)
```

Listing 1: Model and transformations examples

4 Validation : reaching the limits . . .

Our team is working on Web Services Orchestrations merge [2] [Nemo et al., 2007]. Enterprise use *Web Services Business Process Execution Language* (WSBPEL [OASIS, 2007]) to describe orchestrations. WSBPEL is a huge and complex XML

[1] As mathematical operators still obviously exists in LISP, we refer to our − binary operator using *minus* symbolic name, . . .

[2] ADORE Project : `http://rainbow.i3s.unice.fr/adore`

dialect. We define a simplified orchestration language called Boa to foster prototyping of abstract orchestrations.

Boa–defined orchestrations have to be transformed into *(i)* Prolog facts (merging), *(ii)* `C#` code (execution) and *(iii)* `GraphViz` description (display).

`Boac`, a DSL to *s–expr* compiler : We implement a Dsl compiler supporting Boa meta–model (available on the project website). This compiler is written using a functional language, and produces unevaluated *s–expressions*. We implement needed transformations using previously shown technique, by defining functions. It results in a simple compiler source code and really concise and readable transformations expressions.

Discussion : The Lisp evaluation mechanism use an *depth–first* evaluation of *s–expression*, which seems to be a handicap for some meta–models. But, definition of *macros* or *continuations* can be used to handle evaluation and solve this problem.

Short and *elegant* code is more easy to understand than long and complex code. Following the well-known *"Small is beautiful"* proverb, we decide to avoid *Swiss Army knifes* function definitions. For example, we implement variable declaration validation, type consistency checking and *design–time defects* detection as three different set of functions, sequentially evaluated. This approach is not optimal in performance terms, but ensures extensibility and maintainability.

As `boac` is a young experiment, we do not address model evolution and tracking concerns for now.

5 Conclusion

In this paper, we show that some meta–models can be defined using functional concepts, and then transformed using native evaluation mechanisms. We also validate our idea on a real–life software, ready to use rapidly and available for download on the project website.

A lot of meta–models are in facts languages abstraction. *Microsoft* choose to see *Model Driven Development* as *Language Driven Development*. In these cases, functional representations seem to be a good way to express pivot meta–model and to express different transformations. Functional languages answer criteria listed in [Oldevik et al., 2004] about *Model to Text* transformations needs.

Functional mechanisms can express Mda concepts, and are closer than people usually think. We defend that a functional approach of model transformation may fill the gap between those two worlds. Obvious limits and criticism can easily be handled by Lisp native mechanism, relying on a solid and well-known basis. As Lisp initial concepts engender a lot of object features (Clos, dynamic typing, ...), it can maybe be used to enrich Mda features, as a virtual machine for model interpretation.

References

[Gamma et al., 1993] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1993). Design patterns: Abstraction and reuse of object-oriented design. *Lecture Notes in Computer Science*, 707:406–431.

[Klint et al., 2005] Klint, P., Lammel, R., and Verhoef, C. (2005). Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380.

[Muller et al., 2005] Muller, P.-A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., and Jézéquel, J.-M. (2005). On Executable Meta-Languages applied to Model Transformations. In *Model Transformations In Practice Workshop*.

[Muller and Hassenforder, 2005] Muller, P.-A. and Hassenforder, M. (2005). HUTN as a Bridge between ModelWare and GrammarWare . In *WISME Workshop, MODELS / UML'2005 (WISME)*, Montego Bay, Jamaica.

[Nemo et al., 2007] Nemo, C., Blay-Fornarino, M., Kniesel, G., and Riveill, M. (2007). SEMANTIC ORCHESTRATIONS MERGING - Towards Composition of Overlapping Orchestrations. In Filipe, J., editor, *9th International Conference on Enterprise Information Systems (ICEIS'2007)*, Funchal, Madeira.

[OASIS, 2007] OASIS (2007). Web services business process execution language version 2.0. Technical report, OASIS.

[Oldevik et al., 2004] Oldevik, J., Neple, T., and Aagedal, J. O. (2004). Model Abstraction versus Model to Text Transformation. In *Second European Workshop on Model Driven Architecture (MDA)*. University of Kent.

[OMG, 2005] OMG (2005). Model driven architecture official website. http://www.omg.org/mda/.